IN PARTNERSHIP WITH:
CNRS

Sorbonne Université (UPMC)

Activity Report 2019

Project-Team WHISPER

Well Honed Infrastructure Software for Programming Environments and Runtimes

IN COLLABORATION WITH: Laboratoire d'informatique de Paris 6 (LIP6)

RESEARCH CENTER
Paris

THEME
Distributed Systems and middleware

Table of contents

Project-Team WHISPER

Creation of the Team: 2014 May 15, updated into Project-Team: 2015 December 01

Keywords:

Computer Science and Digital Science:

A1. - Architectures, systems and networks
A1.1.1. - Multicore, Manycore
A1.1.13. - Virtualization
A2.1.6. - Concurrent programming
A2.1.10. - Domain-specific languages
A2.1.11. - Proof languages
A2.2.1. - Static analysis
A2.2.5. - Run-time systems
A2.2.8. - Code generation
A2.3.1. - Embedded systems
A2.3.3. - Real-time systems
A2.4. - Formal method for verification, reliability, certification
A2.4.3. - Proofs
A2.5. - Software engineering
A2.6.1. - Operating systems
A2.6.2. - Middleware
A2.6.3. - Virtual machines

Other Research Topics and Application Domains:

B5. - Industry of the future
B5.2.1. - Road vehicles
B5.2.3. - Aviation
B5.2.4. - Aerospace
B6.1. - Software industry
B6.1.1. - Software engineering
B6.1.2. - Software evolution, maintenance
B6.3.3. - Network Management
B6.5. - Information systems
B6.6. - Embedded systems

1. Team, Visitors, External Collaborators

Research Scientists

Gilles Muller [Team leader, Inria, Senior Researcher, HDR]
Pierre-Évariste Dagand [CNRS, Researcher]
Julia Lawall [Inria, Senior Researcher]

Faculty Members

Bertil Folliot [Univ Pierre et Marie Curie, Professor, HDR]
Maria Virginia Aponte Garcia [CNAM, Associate Professor, from Oct 2019]

PhD Students

Cédric Courtaud [Thales, PhD Student, until Jun 2019, granted by CIFRE]
Yoann Ghigoff [Orange Labs, PhD Student, from Jun 2019]
Redha Gouicem [Univ Pierre et Marie Curie, PhD Student]
Darius Mercadier [Univ Pierre et Marie Curie, PhD Student]
Pierre Nigron [Inria, PhD Student, from Nov 2019]
Lucas Serrano [Univ Pierre et Marie Curie, PhD Student]

Technical staff

Michael Damien Carver [Univ Pierre et Marie Curie, Engineer, from Mar 2019]

Administrative Assistants

Nelly Maloisel [Inria, Administrative Assistant]
Eugène Kamdem [SU, Assistant]

2. Overall Objectives

2.1. Overall Objectives

The focus of Whisper is on how to develop (new) and improve (existing) infrastructure software. Infrastructure software (also called systems software) is the software that underlies all computing. Such software allows applications to access resources and provides essential services such as memory management, synchronization and inter-process interactions. Starting bottom-up from the hardware, examples include virtual machine hypervisors, operating systems, managed runtime environments, standard libraries, and browsers, which amount to the new operating system layer for Internet applications. For such software, efficiency and correctness are fundamental. Any overhead will impact the performance of all supported applications. Any failure will prevent the supported applications from running correctly. Since computing now pervades our society, with few paper backup solutions, correctness of software at all levels is critical. Formal methods are increasingly being applied to operating systems code in the research community [44], [50], [79]. Still, such efforts require a huge amount of manpower and a high degree of expertise which makes this work difficult to replicate in standard infrastructure-software development.

In terms of methodology, Whisper is at the interface of the domains of operating systems, software engineering and programming languages. Our approach is to combine the study of problems in the development of real-world infrastructure software with concepts in programming language design and implementation, *e.g.*, of domain-specific languages, and knowledge of low-level system behavior. A focus of our work is on providing support for legacy code, while taking the needs and competences of ordinary system developers into account.

We aim at providing solutions that can be easily learned and adopted by system developers in the short term. Such solutions can be tools, such as Coccinelle [2], [9], [10] for transforming C programs, or domain-specific languages such as Devil [8] and Bossa [7] for designing drivers and kernel schedulers. Due to the small size of the team, Whisper mainly targets operating system kernels and runtimes for programming languages. We put an emphasis on achieving measurable improvements in performance and safety in practice, and on feeding these improvements back to the infrastructure software developer community.

3. Research Program

3.1. Scientific Foundations

3.1.1. Program analysis

A fundamental goal of the research in the Whisper team is to elicit and exploit the knowledge found in existing code. To do this in a way that scales to a large code base, systematic methods are needed to infer

code properties. We may build on either static [36], [38], [40] or dynamic analysis [58], [60], [65]. Static analysis consists of approximating the behavior of the source code from the source code alone, while dynamic analysis draws conclusions from observations of sample executions, typically of test cases. While dynamic analysis can be more accurate, because it has access to information about actual program behavior, obtaining adequate test cases is difficult. This difficulty is compounded for infrastructure software, where many, often obscure, cases must be handled, and external effects such as timing can have a significant impact. Thus, we expect to primarily use static analyses. Static analyses come in a range of flavors, varying in the extent to which the analysis is *sound*, *i.e.*, the extent to which the results are guaranteed to reflect possible run-time behaviors.

One form of sound static analysis is *abstract interpretation* [38]. In abstract interpretation, atomic terms are interpreted as sound abstractions of their values, and operators are interpreted as functions that soundly manipulate these abstract values. The analysis is then performed by interpreting the program in a compositional manner using these abstracted values and operators. Alternatively, *dataflow analysis* [49] iteratively infers connections between variable definitions and uses, in terms of local transition rules that describe how various kinds of program constructs may impact variable values. Schmidt has explored the relationship between abstract interpretation and dataflow analysis [73]. More recently, more general forms of symbolic execution [36] have emerged as a means of understanding complex code. In symbolic execution, concrete values are used when available, and these are complemented by constraints that are inferred from terms for which only partial information is available. Reasoning about these constraints is then used to prune infeasible paths, and obtain more precise results. A number of works apply symbolic execution to operating systems code [33], [34].

While sound approaches are guaranteed to give correct results, they typically do not scale to the very diverse code bases that are prevalent in infrastructure software. An important insight of Engler et al. [42] was that valuable information could be obtained even when sacrificing soundness, and that sacrificing soundness could make it possible to treat software at the scales of the kernels of the Linux or BSD operating systems. Indeed, for certain types of problems, on certain code bases, that may mostly follow certain coding conventions, it may mostly be safe to *e.g.*, ignore the effects of aliases, assume that variable values are unchanged by calls to unanalyzed functions, etc. Real code has to be understood by developers and thus cannot be too complicated, so such simplifying assumptions are likely to hold in practice. Nevertheless, approaches that sacrifice soundness also require the user to manually validate the results. Still, it is likely to be much more efficient for the user to perform a potentially complex manual analysis in a specific case, rather than to implement all possible required analyses and apply them everywhere in the code base. A refinement of unsound analysis is the CEGAR approach [37], in which a highly approximate analysis is complemented by a sound analysis that checks the individual reports of the approximate analysis, and then any errors in reasoning detected by the sound analysis are used to refine the approximate analysis. The CEGAR approach has been applied effectively on device driver code in tools developed at Microsoft [25]. The environment in which the driver executes, however, is still represented by possibly unsound approximations.

Going further in the direction of sacrificing soundness for scalability, the software engineering community has recently explored a number of approaches to code understanding based on techniques developed in the areas of natural language understanding, data mining, and information retrieval. These approaches view code, as well as other software-reated artifacts, such as documentation and postings on mailing lists, as bags of words structured in various ways. Statistical methods are then used to collect words or phrases that seem to be highly correlated, independently of the semantics of the program constructs that connect them. The obliviousness to program semantics can lead to many false positives (invalid conclusions) [55], but can also highlight trends that are not apparent at the low level of individual program statements. We have previously explored combining such statistical methods with more traditional static analysis in identifying faults in the usage of constants in Linux kernel code [53].

3.1.2. Domain Specific Languages

Writing low-level infrastructure code is tedious and difficult, and verifying it is even more so. To produce non-trivial programs, we could benefit from moving up the abstraction stack to enable both programming and

proving as quickly as possible. Domain-specific languages (DSLs), also known as *little languages*, are a means to that end [6] [61].

3.1.2.1. Traditional approach.

Using little languages to aid in software development is a tried-and-trusted technique [75] by which programmers can express high-level ideas about the system at hand and avoid writing large quantities of formulaic C boilerplate.

This approach is typified by the Devil language for hardware access [8]. An OS programmer describes the register set of a hardware device in the high-level Devil language, which is then compiled into a library providing C functions to read and write values from the device registers. In doing so, Devil frees the programmer from having to write extensive bit-manipulation macros or inline functions to map between the values the OS code deals with, and the bit-representation used by the hardware: Devil generates code to do this automatically.

However, DSLs are not restricted to being "stub" compilers from declarative specifications. The Bossa language [7] is a prime example of a DSL involving imperative code (syntactically close to C) while offering a high-level of abstraction. This design of Bossa enables the developer to implement new process scheduling policies at a level of abstraction tailored to the application domain.

Conceptually, a DSL both abstracts away low-level details and justifies the abstraction by its semantics. In principle, it reduces development time by allowing the programmer to focus on high-level abstractions. The programmer needs to write less code, in a language with syntax and type checks adapted to the problem at hand, thus reducing the likelihood of errors.

3.1.2.2. Embedding DSLs.

The idea of a DSL has yet to realize its full potential in the OS community. Indeed, with the notable exception of interface definition languages for remote procedure call (RPC) stubs, most OS code is still written in a low-level language, such as C. Where DSL code generators are used in an OS, they tend to be extremely simple in both syntax and semantics. We conjecture that the effort to implement a given DSL usually outweighs its benefit. We identify several serious obstacles to using DSLs to build a modern OS: specifying what the generated code will look like, evolving the DSL over time, debugging generated code, implementing a bug-free code generator, and testing the DSL compiler.

Filet-o-Fish (FoF) [39] addresses these issues by providing a framework in which to build correct code generators from semantic specifications. This framework is presented as a Haskell library, enabling DSL writers to *embed* their languages within Haskell. DSL compilers built using FoF are quick to write, simple, and compact, but encode rigorous semantics for the generated code. They allow formal proofs of the run-time behavior of generated code, and automated testing of the code generator based on randomized inputs, providing greater test coverage than is usually feasible in a DSL. The use of FoF results in DSL compilers that OS developers can quickly implement and evolve, and that generate provably correct code. FoF has been used to build a number of domain-specific languages used in Barrelfish, [26] an OS for heterogeneous multicore systems developed at ETH Zurich.

The development of an embedded DSL requires a few supporting abstractions in the host programming language. FoF was developed in the purely functional language Haskell, thus benefiting from the type class mechanism for overloading, a flexible parser offering convenient syntactic sugar, and purity enabling a more algebraic approach based on small, composable combinators. Object-oriented languages – such as Smalltalk [43] and its descendant Pharo [30] – or multi-paradigm languages – such as the Scala programming language [63] – also offer a wide range of mechanisms enabling the development of embedded DSLs. Perhaps suprisingly, a low-level imperative language – such as C – can also be extended so as to enable the development of embedded compilers [27].

3.1.2.3. Certifying DSLs.

Whilst automated and interactive software verification tools are progressively being applied to larger and larger programs, we have not yet reached the point where large-scale, legacy software – such as the Linux kernel –

could formally be proved "correct". DSLs enable a pragmatic approach, by which one could realistically strengthen a large legacy software by first narrowing down its critical component(s) and then focus our verification efforts onto these components.

Dependently-typed languages, such as Coq or Idris, offer an ideal environment for embedding DSLs [35], [31] in a unified framework enabling verification. Dependent types support the type-safe embedding of object languages and Coq's mixfix notation system enables reasonably idiomatic domain-specific concrete syntax. Coq's powerful abstraction facilities provide a flexible framework in which to not only implement and verify a range of domain-specific compilers [39], but also to combine them, and reason about their combination.

Working with many DSLs optimizes the "horizontal" compositionality of systems, and favors reuse of building blocks, by contrast with the "vertical" composition of the traditional compiler pipeline, involving a stack of comparatively large intermediate languages that are harder to reuse the higher one goes. The idea of building compilers from reusable building blocks is a common one, of course. But the interface contracts of such blocks tend to be complex, so combinations are hard to get right. We believe that being able to write and verify formal specifications for the pieces will make it possible to know when components can be combined, and should help in designing good interfaces.

Furthermore, the fact that Coq is also a system for formalizing mathematics enables one to establish a close, formal connection between embedded DSLs and non-trivial domain-specific models. The possibility of developing software in a truly "model-driven" way is an exciting one. Following this methodology, we have implemented a certified compiler from regular expressions to x86 machine code [48]. Interestingly, our development crucially relied on an existing Coq formalization, due to Braibant and Pous, [32] of the theory of Kleene algebras.

While these individual experiments seem to converge toward embedding domain-specific languages in rich type theories, further experimental validation is required. Indeed, Barrelfish is an extremely small software compared to the Linux kernel. The challenge lies in scaling this methodology up to large software systems. Doing so calls for a unified platform enabling the development of a myriad of DSLs, supporting code reuse across DSLs as well as providing support for mechanically-verified proofs.

3.2. Research direction: Tools for improving legacy infrastructure software

A cornerstone of our work on legacy infrastructure software is the Coccinelle program matching and transformation tool for C code. Coccinelle has been in continuous development since 2005. Today, Coccinelle is extensively used in the context of Linux kernel development, as well as in the development of other software, such as wine, python, kvm, and systemd. Currently, Coccinelle is a mature software project, and no research is being conducted on Coccinelle itself. Instead, we leverage Coccinelle in other research projects [28], [29], [64], [66], [70], [72], [74], [59], [54], both for code exploration, to better understand at a large scale problems in Linux development, and as an essential component in tools that require program matching and transformation. The continuing development and use of Coccinelle is also a source of visibility in the Linux kernel developer community. We submitted the first patches to the Linux kernel based on Coccinelle in 2007. Since then, over 5500 patches have been accepted into the Linux kernel based on the use of Coccinelle, including around 3000 by over 500 developers from outside our research group.

Our recent work has focused on driver porting. Specifically, we have considered the problem of porting a Linux device driver across versions, particularly backporting, in which a modern driver needs to be used by a client who, typically for reasons of stability, is not able to update their Linux kernel to the most recent version. When multiple drivers need to be backported, they typically need many common changes, suggesting that Coccinelle could be applicable. Using Coccinelle, however, requires writing backporting transformation rules. In order to more fully automate the backporting (or symmetrically forward porting) process, these rules should be generated automatically. We have carried out a preliminary study in this direction with David Lo of Singapore Management University; this work, published at ICSME 2016 [77], is limited to a port from one version to the next one, in the case where the amount of change required is limited to a single line of code. Whisper has been awarded an ANR PRCI grant to collaborate with the group of David Lo on scaling up the rule inference process and proposing a fully automatic porting solution.

3.3. Research direction: developing infrastructure software using Domain Specific Languages

We wish to pursue a *declarative* approach to developing infrastructure software. Indeed, there exists a significant gap between the high-level objectives of these systems and their implementation in low-level, imperative programming languages. To bridge that gap, we propose an approach based on domain-specific languages (DSLs). By abstracting away boilerplate code, DSLs increase the productivity of systems programmers. By providing a more declarative language, DSLs reduce the complexity of code, thus the likelihood of bugs.

Traditionally, systems are built by accretion of several, independent DSLs. For example, one might use Devil [8] to interact with devices, Bossa [7] to implement the scheduling policies. However, much effort is duplicated in implementing the back-ends of the individual DSLs. Our long term goal is to design a unified framework for developing and composing DSLs, following our work on Filet-o-Fish [39]. By providing a single conceptual framework, we hope to amortize the development cost of a myriad of DSLs through a principled approach to reusing and composing them.

Beyond the software engineering aspects, a unified platform brings us closer to the implementation of mechanically-verified DSLs. Using the Coq proof assistant as an x86 macro-assembler [48] is a step in that direction, which belongs to a larger trend of hosting DSLs in dependent type theories [31], [35], [62]. A key benefit of those approaches is to provide – by construction – a formal, mechanized semantics to the DSLs thus developed. This semantics offers a foundation on which to base further verification efforts, whilst allowing interaction with non-verified code. We advocate a methodology based on incremental, piece-wise verification. Whilst building fully-certified systems from the top-down is a worthwhile endeavor [50], we wish to explore a bottom-up approach by which one focuses first and foremost on crucial subsystems and their associated properties.

Our current work on DSLs has two complementary goals: (i) the design of a unified framework for developing and composing DSLs, following our work on Filet-o-Fish, and (ii) the design of domain-specific languages for domains where there is a critical need for code correctness, and corresponding methodologies for proving properties of the run-time behavior of the system.

4. Application Domains

4.1. Linux

Linux is an open-source operating system that is used in settings ranging from embedded systems to supercomputers. The most recent release of the Linux kernel, v4.14, comprises over 16 million lines of code, and supports 30 different families of CPU architectures, around 50 file systems, and thousands of device drivers. Linux is also in a rapid stage of development, with new versions being released roughly every 2.5 months. Recent versions have each incorporated around 13,500 commits, from around 1500 developers. These developers have a wide range of expertise, with some providing hundreds of patches per release, while others have contributed only one. Overall, the Linux kernel is critical software, but software in which the quality of the developed source code is highly variable. These features, combined with the fact that the Linux community is open to contributions and to the use of tools, make the Linux kernel an attractive target for software researchers. Tools that result from research can be directly integrated into the development of real software, where it can have a high, visible impact.

Starting from the work of Engler et al. [41], numerous research tools have been applied to the Linux kernel, typically for finding bugs [40], [57], [67], [76] or for computing software metrics [46], [78]. In our work, we have studied generic C bugs in Linux code [10], bugs in function protocol usage [51], [52], issues related to the processing of bug reports [71] and crash dumps [45], and the problem of backporting [66], [77], illustrating the variety of issues that can be explored on this code base. Unique among research groups working in this area, we have furthermore developed numerous contacts in the Linux developer community. These contacts provide insights into the problems actually faced by developers and serve as a means of validating the practical relevance of our work.

4.2. Device Drivers

Device drivers are essential to modern computing, to provide applications with access, via the operating system, to physical devices such as keyboards, disks, networks, and cameras. Development of new computing paradigms, such as the internet of things, is hampered because device driver development is challenging and error-prone, requiring a high level of expertise in both the targeted OS and the specific device. Furthermore, implementing just one driver is often not sufficient; today's computing landscape is characterized by a number of OSes, *e.g.*, Linux, Windows, MacOS, BSD and many real time OSes, and each is found in a wide range of variants and versions. All of these factors make the development, porting, backporting, and maintenance of device drivers a critical problem for device manufacturers, industry that requires specific devices, and even for ordinary users.

The last fifteen years have seen a number of approaches directed towards easing device driver development. Réveillère, who was supervised by G. Muller, proposes Devil [8], a domain-specific language for describing the low-level interface of a device. Chipounov *et al.* propose RevNic, [34] a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite, [68], [69] an approach for synthesizing device driver code from a specification of an OS and a device. Currently, these approaches have been successfully applied to only a small number of toy drivers. Indeed, Kadav and Swift [47] observe that these approaches make assumptions that are not satisfied by many drivers; for example, the assumption that a driver involves little computation other than the direct interaction between the OS and the device. At the same time, a number of tools have been developed for finding bugs in driver code. These tools include SDV [25], Coverity [41], CP-Miner, [56] PR-Miner [57], and Coccinelle [9]. These approaches, however, focus on analyzing existing code, and do not provide guidelines on structuring drivers.

In summary, there is still a need for a methodology that first helps the developer understand the software architecture of drivers for commonly used operating systems, and then provides tools for the maintenance of existing drivers.

5. Highlights of the Year

5.1. Highlights of the Year

The Whisper team published one paper at USENIX ATC, one at ASPLOS, one at EuroSys, one at RTSS, and one at PLDI which are five of the major conferences in the scope of our team:

- Effective Static Analysis of Concurrency Use-After-FreeBugs in Linux Device Drivers. USENIX 2019. [13]
- DCNS: Automated Detection of Conservative Non-SleepDefects in the Linux Kernel. ASPLOS 2019. [14]
- When eXtended Para-Virtualization (XPV) meets NUMA. EuroSys 2019. [15]
- Improving Prediction Accuracy of Memory Interferences for Multicore Platforms, RTSS 2019 [18]
- Usuba: high-throughput and constant-time ciphers, by construction. PLDI 2019. [23]

Julia Lawall was co-PC chair of the ASE 2019 research paper track.

6. New Software and Platforms

6.1. Coccinelle

KEYWORDS: Code quality - Evolution - Infrastructure software

FUNCTIONAL DESCRIPTION: Coccinelle is a tool for code search and transformation for C programs. It has been extensively used for bug finding and evolutions in Linux kernel code.

- Participants: Gilles Muller, Julia Lawall, Nicolas Palix, Rene Rydhof Hansen and Thierry Martinez
- Partners: LIP6 - IRILL
- Contact: Julia Lawall
- URL: http://coccinelle.lip6.fr

6.2. Prequel

KEYWORDS: Code search - Git

SCIENTIFIC DESCRIPTION: The commit history of a code base such as the Linux kernel is a gold mine of information on how evolutions should be made, how bugs should be fixed, etc. Nevertheless, the high volume of commits available and the rudimentary filtering tools provided mean that it is often necessary to wade through a lot of irrelevant information before finding example commits that can help with a specific software development problem. To address this issue, we propose Prequel (Patch Query Language), which brings the descriptive power of code matching to the problem of querying a commit history.

FUNCTIONAL DESCRIPTION: Prequel is a tool for searching for complex patterns in the commits of software managed using git.

- Participants: Gilles Muller and Julia Lawall
- Partners: LIP6 - IRILL
- Contact: Julia Lawall
- URL: http://prequel-pql.gforge.inria.fr/

6.3. Usuba

KEYWORDS: Cryptography - Optimizing compiler - Synchronous Language

FUNCTIONAL DESCRIPTION: Usuba is a programming language for specifying block ciphers as well as a bitslicing compiler, for producing high-throughput and secure code.

- Contact: Pierre-Evariste Dagand
- Publication: Usuba, Optimizing & Trustworthy Bitslicing Compiler
- URL: https://github.com/DadaIsCrazy/usuba/

6.4. SchedDisplay

KEYWORDS: Linux kernel - Scheduling - Multicore

FUNCTIONAL DESCRIPTION: SchedDisplay is a visualization tool for SchedLog, a custom ring buffer collecting scheduling events in the Linux kernel. SchedDisplay allows kernel developers to analyze the behavior of the Linux scheduler while running a multicore application.

RELEASE FUNCTIONAL DESCRIPTION: First version released as part of a Demo made during the 10th PLOS workshop: https://ess.cs.uni-osnabrueck.de/workshops/plos/2019/program.php

- Partner: Oracle Labs
- Contact: Gilles Muller
- URL: https://gitlab.inria.fr/gmuller/scheddisplay

7. New Results

7.1. Software engineering for infrastructure software

Data races are often hard to detect in device drivers, due to the non-determinism of concurrent execution. With colleagues from Tsinghua University, we have addressed this issue using dynamic analysis. According to our study of Linux driver patches that fix data races, more than 38% of patches involve a pattern that we call inconsistent lock protection. Specifically, if a variable is accessed within two concurrently executed functions, the sets of locks held aroundeach access are disjoint, at least one of the locksets is non-empty, and at least one of the involved accesses is a write, then a datarace may occur. In a paper published at SANER 2019 [17], we present a runtime analysis approach, named DILP, to detect data races caused by inconsistent lock protection in device drivers. By monitoring driver execution, DILP collects the information about runtime variable accesses and executed functions. Then after driver execution, DILP analyzes the collected information to detect and report data races caused by inconsistent lock protection. We evaluate DILP on 12 device drivers in Linux 4.16.9, and find 25 real data races.

For waiting, the Linux kernel offers both sleep-able and non-sleep operations. However, only non-sleep operations can be used in atomic context. Detecting the possibility of execution in atomic context requires a complete inter-procedural flow analysis, often involving function pointers. Developers may thus conservatively use non-sleep operations even outside of atomic context, which may damage system performance, as such operations unproductively monopolize the CPU. Until now, no systematic approach has been proposed to detect such conservative non-sleep (CNS) defects. In a paper published at ASPLOS 2019 [14] with colleagues from Tsinghua University, we propose a practical static approach, named DCNS, to automatically detect conservative non-sleep defects in the Linux kernel. DCNS uses a summary-based analysis to effectively identify the code in atomic context and a novel file-connection-based alias analysis to correctly identify the set of functions referenced by a function pointer. We evaluate DCNS on Linux 4.16, and in total find 1629 defects. We manually check 943 defects whose call paths are not so difficult to follow, and find that 890 are real. We have randomly selected 300 of the real defects and sent them to kernel developers, and 251 have been confirmed.

In Linux device drivers, use-after-free (UAF) bugs can cause system crashes and serious security problems. We have addressed this issue in work with colleagues at Tsinghua University. According to our study of Linux kernel commits, 42% of the driver commits fixing use-after-free bugs involve driver concurrency. We refer to these use-after-free bugs as concurrency use-after-free bugs. Due to the non-determinism of concurrent execution, concurrency use-after-free bugs are often more difficult to reproduce and detect than sequential use-after-free bugs. In a paper published at USENIX ATC 2019 [13], we propose a practical static analysis approach named DCUAF, to effectively detect concurrency use-after-free bugs in Linux device drivers. DCUAF combines a local analysis analyzing the source code of each driver with a global analysis statistically analyzing the local results of all drivers, forming a local-global analysis, to extract the pairs of driver interface functions that may be concurrently executed. Then, with these pairs, DCUAF performs a summary-based lockset analysis to detect concurrency use-after-free bugs. We have evaluated DCUAF on the driver code of Linux 4.19, and found 640 real concurrency use-after-free bugs. We have randomly selected 130 of the real bugs and reported them to Linux kernel developers, and 95 have been confirmed.

Linux kernel stable versions serve the needs of users who value stability of the kernel over new features. The quality of such stableversions depends on the initiative of kernel developers and maintainers to propagate bug fixing patches to the stable versions. Thus, it is desirable to consider to what extent this process can be automated. A previous approach relies on words from commit messages and a small set of manually constructed code features. This approach, however, shows only moderate accuracy. In a tool paper published ICSE 2019 [11], in the context of the ANR-NRF ITrans project with colleagues from Singapore Management University, paper, we investigate whether deep learning can provide a more accurate solution. We propose PatchNet, a hierarchical deep learning-based approach capable of automatically extracting features from commit messages and commit code and usingthem to identify stable patches. PatchNet contains a deep

hierarchical structure that mirrors the hierarchical and sequential structure of commit code, making it distinctive from the existing deep learning models on source code. Experiments on 82,403 recent Linux patches confirm the superiority of PatchNet against various state-of-the-art baselines, including the one recently-adopted by Linux kernel maintainers.

Developing software often requires code changes that are widespread and applied to multiple locations. Previously, the Whisper team has addressed this problem with the tool Coccinelle. In a recent experience paper, published at ECOOP 2019 [21], in the context of the ANR-NRF ITrans project with colleagues from Singapore Management University, we have considered the benefits of extending Coccinelle to Java code. There are tools for Java that allow developers to specify patterns for program matching and source-to-source transformation. However, to our knowledge, none allows for transforming code based on its control-flow context. We prototype Coccinelle4J, an extension to Coccinelle, which is a program transformation tool designed for widespread changes in C code, in order to work on Java source code. We adapt Coccinelle to be able to apply scripts written in the Semantic Patch Language (SmPL), a language provided by Coccinelle, to Java source files. As a case study, we demonstrate the utility of Coccinelle4J with the task of API migration. We show 6 semantic patches to migrate from deprecated Android API methods on several open source Android projects. We describe how SmPL can be used to express several API migrations and justify several of our design decisions. This paper was accompanied by a tool demo.

A challenge in designing cooperative distributed systems is to develop feasible and cost-effective mechanisms to foster cooperation among selfish nodes, i.e., nodes that strategically deviate from the intended specification to increase their individual utility. Finding a satisfactory solution to this challenge may be complicated by the intrinsic characteristics of each system, as well as by the particular objectives set by the system designer. In a previous work we addressed this challenge by proposing RACOON, a general and semi-automatic framework for designing selfishness-resilient cooperative systems. RACOON relies on classical game theory and a custom built simulator to predict the impact of a fixed set of selfish behaviours on the designer's objectives. In a paper published in IEEE Transactions on Dependable and Secure Computing [12], we present RACOON++, which extends the previous framework with a declarative model for defining the utility function and the static behaviour of selfish nodes, along with a new model for reasoning on the dynamic interactions of nodes, based on evolutionary game theory. We illustrate the benefits of using RACOON++ by designing three cooperative systems: a peer-to-peer live streaming system, a load balancing protocol, and an anonymous communication system. Extensive experimental results using the state-of-the-art PeerSim simulator verify that the systems designed using RACOON++ achieve both selfishness-resilience and high performance.

7.2. Programming after the end of Moore's law

The end of Moore's law is a wake-up call that resonates across Computer Science at large. We are now firmly in an era of custom hardware design, as witnessed by the diversity of system-on-chip (SoC) and specialized processing units – such as graphics processing units (GPUs), tensor processing unit (TPUs) or programmable network adapters, to name but a few. This trend is justified by the existence of niche application domains (graphic processing, linear algebra, packet processing, etc.) that greatly benefit from specialized hardware. Faced with the imminent explosion of the number of niche applications and niche architectures, we are still grasping for a programming model that would accommodate this diversity.

The Usuba project is an exploratory effort in that direction. We chose a niche application domain (symmetric cryptographic algorithms), a specialized execution platform (Single Instruction Multiple Data, SIMD) processors and we set out to design a programming language faithfully describing our application domain as well as an optimizing compiler efficiently exploiting our target execution platform.

Indeed, cryptographic primitives are subject to diverging imperatives. Functional correctness and auditability pushes for the use of a high-level programming language. Performance and the threat of timing attacks push for directly programming in assembler to exploit (or avoid!) the micro-architectural features of a given machine. In a paper published at PLDI 2019 [23], we have demonstrated that a suitable programming language could reconcile both views and actually improve on the state of the art of both.

USUBA is a dataflow programming language in which block ciphers become so simple as to be "obviously correct" and whose types document and enforce valid parallelization strategies at the granularity of individual bits. Its optimizing compiler, USUBAC, produces high-throughput, constant-time implementations performing on par with hand-tuned reference implementations. The cornerstone of our approach is a systematization and generalization of *bitslicing*, an implementation trick frequently used by cryptographers. We have shown that USUBA can produce code that executes between 5% slower to 22% faster than hand-tuned reference implementations while gracefully scaling across a wide range of architectures and automatically exploiting Single Instruction Multiple Data (SIMD) instructions whenever the cipher's structure allows it.

7.3. Support for multicore machines

The complexity of computer architectures has risen since the early years of the Linux kernel: Simultaneous Multi-Threading (SMT), multicore processing, and frequency scaling with complex algorithms such as Intel Turbo Boost have all become omnipresent. In order to keep up with hardware innovations, the Linux scheduler has been rewritten several times, and many hardware-related heuristics have been added. Despite this, we have shown in a PLOS paper [16] that a fundamental problem was never identified: the POSIX process creation model, i.e., fork/wait, can behave inefficiently on current multicore architectures due to frequency scaling. We investigate this issue through a simple case study: the compilation of the Linux kernel source tree. To do this, we have developed SchedLog, a low-overhead scheduler tracing tool, and SchedDisplay, a scriptable tool to graphically analyze SchedLog's traces efficiently. We implement two solutions to the problem at the scheduler level which improve the speed of compiling part of the Linux kernel by up to 26%, and the whole kernel by up to 10%.

In an Eurosys paper [15], we address the problem of efficiently virtualizing NUMA architectures. The major challenge comes from the fact that the hypervisor regularly reconfigures the placement of a virtual machine (VM) over the NUMA topology. However, neither guest operating systems (OSes) nor system runtime libraries (e.g., Hotspot) are designed to consider NUMA topology changes at runtime, leading end user applications to unpredictable performance. We present eXtended Para-Virtualization (XPV), a new principle to efficiently virtualize a NUMA architecture. XPV consists in revisiting the interface between the hypervisor and the guest OS, and between the guest OS and system runtime libraries (SRL) so that they can dynamically take into account NUMA topology changes. We introduce a methodology for systematically adapting legacy hypervisors, OSes, and SRLs. We have applied our approach with less than 2k line of codes in two legacy hypervisors (Xen and KVM), two legacy guest OSes (Linux and FreeBSD), and three legacy SRLs (Hotspot, TCMalloc, and jemalloc). The evaluation results showed that XPV outperforms all existing solutions by up to 304%.

Memory interferences may introduce important slowdowns in applications running on COTS multi-core processors. They are caused by concurrent accesses to shared hardware resources of the memory system. The induced delays are difficult to predict, making memory interferences a major obstacle to the adoption of COTS multi-core processors in real-time systems. In an RTSS paper[18], we propose an experimental characterization of applications' memory consumption to determine their sensitivity to memory interferences. Thanks to a new set of microbenchmarks, we show the lack of precision of a purely quantitative characterization. To improve accuracy, we define new metrics quantifying qualitative aspects of memory consumption and implement a profiling tool using the VALGRIND framework. In addition, our profiling tool produces high resolution profiles allowing us to clearly distinguish the various phases in applications' behavior. Using our microbenchmarks and our new characterization, we train a state-of-the-art regressor. The validation on applications from the MIBENCH and the PARSEC suites indicates significant gain in prediction accuracy compared to a purely quantitative characterization.

8. Bilateral Contracts and Grants with Industry

8.1. Bilateral Contracts with Industry

- Orange Labs, 2019-2021, 30 000 euros. The purpose of this contract is to design application-specific proxies so as to speed up network services. The PhD of Yoann Ghigoff is supported by a CIFRE fellowship as part of this contract.
- Thales Research, 2016-2019, 45 000 euros. The purpose of this contract is to enable the usage of multicore architectures in avionics systems. The PhD of Cédric Courtaud is supported by a CIFRE fellowship as part of this contract.
- DGA-Inria, 2019-2021, 60 000 euros. The purpose of this PhD grant is to develop a high-performance, certified packet processing system. The PhD of Pierre Nigron is supported by this grant.

8.2. Bilateral Grants with Industry

- Oracle, 2018-2019, 100 000 dollars.

Operating system schedulers are often a performance bottleneck on multicore architectures because in order to scale, schedulers cannot make optimal decisions and instead have to rely on heuristics. Detecting that performance degradation comes from the scheduler level is extremely difficult because the issue has not been recognized until recently, and with traditional profilers, both the application and the scheduler affect the monitored metrics in the same way.

The first objective of this project is to produce a profiler that makes it possible to find out whether a bottleneck during application runtime is caused by the application itself, by suboptimal OS scheduler behavior, or by a combination of the two. It will require understanding, analyzing and classifying performance bottlenecks that are caused by schedulers, and devising ways to detect them and to provide enough information for the user to understand the root cause of the issue. Following this, the second objective of this project is to use the profiler to better understand which kinds of workloads suffer from poor scheduling, and to propose new algorithms, heuristics and/or a new scheduler design that will improve the situation. Finally, the third contribution will be a methodology that makes it possible to track scheduling bottlenecks in a specific workload using the profiler, to understand them, and to fix them either at the application or at the scheduler level. We believe that the combination of these three contributions will make it possible to fully harness the power of multicore architectures for any workload.

As part of this project, we have already identified frequency scaling and the “fork/wait” paradigm as a source of inefficiency in modern multicore machines. These first results were published in the PLOS workshop that is held together with SOSP. The diagnosis of the problem was possible thanks to the *SchedLog and* SchedDisplay tools that we developed as part of this project.

9. Partnerships and Cooperations

9.1. Regional Initiatives

- City of Paris, 2016-2019, 100 000 euros. As part of the “Émergence - young team” program the city of Paris is supporting part of our work on domain-specific languages and trustworthy domain-specific compilers.

9.2. National Initiatives

9.2.1. ANR

ITrans - awarded in 2016, duration 2017 - 2020

Members: LIP6 (Whisper), David Lo (Singapore Management University)

Coordinator: Julia Lawall

Whisper members: Julia Lawall, Gilles Muller, Lucas Serrano, Van-Anh Nguyen

Funding: ANR PRCI, 287,820 euros.

Objectives:

Large, real-world software must continually change, to keep up with evolving requirements, fix bugs, and improve performance, maintainability, and security. This rate of change can pose difficulties for clients, whose code cannot always evolve at the same rate. This project will target the problems of *forward porting*, where one software component has to catch up to a code base with which it needs to interact, and *back porting*, in which it is desired to use a more modern component in a context where it is necessary to continue to use a legacy code base, focusing on the context of Linux device drivers. In this project, we will take a *history-guided source-code transformation-based* approach, which automatically traverses the history of the changes made to a software system, to find where changes in the code to be ported are required, gathers examples of the required changes, and generates change rules to incrementally back port or forward port the code. Our approach will be a success if it is able to automatically back and forward port a large number of drivers for the Linux operating system to various earlier and later versions of the Linux kernel with high accuracy while requiring minimal developer effort. This objective is not achievable by existing techniques.

VeriAmos - awarded in 2018, duration 2018 - 2021

Members: Inria (Antique, Whisper), UGA (Erods)

Coordinator: Xavier Rival

Whisper members: Julia Lawall, Gilles Muller

Funding: ANR, 121,739 euros.

Objectives:

General-purpose Operating Systems, such as Linux, are increasingly used to support high-level functionalities in the safety-critical embedded systems industry with usage in automotive, medical and cyber-physical systems. However, it is well known that general purpose OSes suffer from bugs. In the embedded systems context, bugs may have critical consequences, even affecting human life. Recently, some major advances have been done in verifying OS kernels, mostly employing interactive theorem-proving techniques. These works rely on the formalization of the programming language semantics, and of the implementation of a software component, but require significant human intervention to supply the main proof arguments. The VeriAmos project will attack this problem by building on recent advances in the design of domain-specific languages and static analyzers for systems code. We will investigate whether the restricted expressiveness and the higher level of abstraction provided by the use of a DSL will make it possible to design static analyzers that can statically and fully automatically verify important classes of semantic properties on OS code, while retaining adequate performance of the OS service. As a specific use-case, the project will target I/O scheduling components.

9.3. International Initiatives

9.3.1. Inria Associate Teams Not Involved in an Inria International Labs

9.3.1.1. CSG

Title: Proving Concurrent Multi-Core Operating Systems

International Partner (Institution - Laboratory - Researcher):

University of Sydney (Australia) - Willy Zwaenepoel

Start year: 2019

See also: https://team.inria.fr/csgroup/

The initial topic of this cooperation is the development of proved multicore schedulers. Over the last two years, we have explored a novel approach based on the identification of key scheduling abstractions and the realization of these abstractions as a Domain-Specific Language (DSL), Ipanema. We have introduced a concurrency model that relies on execution of scheduling events in mutual execution locally on a core, but that still permits reading the state of other cores without requiring locks.

In the three next years, we will leverage on our existing results towards the following directions: (i) Better understanding of what should be the best scheduler for a given multicore application, (ii) Proving the correctness of the C code generated from the DSL policy and of the Ipanema abstract machine, (iii) Extend the Ipanema DSL to the domain of I/O request scheduling, (iv) Design of a provable complete concurrent kernel.

9.3.2. Inria International Partners

9.3.2.1. Informal International Partners

Julia Lawall and Gilles Muller collaborate with David Lo and Lingxiao Jiang of Singapore Management University in the context of the ANR-NRF funded project ITrans. This project supports the PhD of Lucas Serrano. In 2019, this collaboration led to an experience paper at ECOOP on a transformation tool (a variant of Coccinelle) for Java [21] and a tool paper at ICSE on using machine learning for identifying bug-fixing patches for the Linux kernel [19]. The latter has been extended to a journal article published in the IEEE Transactions on Software Engineering [11]. Lawall and Serrano spent two weeks visiting Lo and Jiang at Singapore Management University in December 2019.

Julia Lawall collaborates with Jia-Ju Bai at Tsinghua University on bug finding for the Linux kernel. In 2019, this collaboration led to a paper at SANER on detecting data races in device drivers [17], a paper at ISSRE on extending Linux kernel fuzzing to be able to detect bugs in error-handling code [20], a paper at ASPLOS on detection of unnecessary spinning in the Linux kernel [14], a paper at USENIX ATC on detection of use-after free concurrency bugs in the Linux kernel [13]. Bai visited the Whisper team for 2 months starting in January 2019. Lawall visited Bai at Tsinghua University for one week in August.

Michele Martone of the Leibniz Supercomputing Centre in Munich, Germany has been using Coccinelle in an HPC context and giving workshops on Coccinelle in the HPC research engineer community. Martone has contributed some patches to Coccinelle and we keep in touch with him about possible improvements to Coccinelle that may have an impact on its use in the HPC community.

9.4. International Research Visitors

9.4.1. Visits of International Scientists

Jia-Ju Bai visited the Whisper team for 2 months starting in January 2019. During this time, he and Julia Lawall worked on a prototype of an interprocedural program analysis tool for C code.

Victor Miraldo (Ultrech University) visited the Whisper team for 2 weeks in June 2019, where he worked with Pierre-Évariste Dagand on data structures for efficient differencing of data structures.

9.4.1.1. Internships

Pierre-Évariste Dagand has supervised the Master 2 research internship of Pierre Nigron (University Paris Diderot), from April to August 2019, on the topic of “Effectful programs and their proofs in a dependently-typed setting”. Pierre Nigron was awarded a DGA-Inria grant to pursue a PhD under Julia Lawall’s supervision, co-supervised by Pierre-Évariste Dagand.

Pierre-Évariste Dagand has supervised the Bachelor research internship of Quentin Corradi (École Normale Supérieure de Lyon), for 6 weeks starting in June 2019, on the topic of “A Formal Semantics of SIMD Instruction Sets”.

Pierre-Évariste Dagand has supervised a pre-doctoral internship of Rémi Oudin (École Normale Supérieure de Cachan), from April to August 2019, on the topic of "Hardware interfaces for transiently-powered systems". Rémi Oudin was awarded a "Contrat Doctoral Spécifique pour Normaliens".

9.4.2. Visits to International Teams

Gilles Muller spent two weeks in November 2019 visiting the University of Sydney as part of our associated team.

Julia Lawall spent one week at Tsinghua University visiting the group of Jia-Ju Bai in August 2019. Julia Lawall and Lucas Serrano spent two weeks at Singapore Management University visiting the group of David Lo and Lingxiao Jiang in December 2019. During the latter visit, Lawall and Serrano also visited National University of Singapore and Lawall also visited Yale-NUS.

10. Dissemination

10.1. Promoting Scientific Activities

10.1.1. Scientific Events: Organisation

10.1.1.1. General Chair, Scientific Chair

Gilles Muller is the elected representative member of the system community in the steering committee of the COMPAS conference and also the head of the committee.

10.1.1.2. Member of the Organizing Committees

Gilles Muller was one the co-organizers of the 10th PLOS Workshop that was associated with SOSP.

10.1.2. Scientific Events: Selection

10.1.2.1. Chair of Conference Program Committees

Julia Lawall was program chair of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019) with Darko Marinov (University of Illinois).

10.1.2.2. Member of the Conference Program Committees

- Julia Lawall: ICSE 2020 New Ideas and Emerging Results, 2019 USENIX Annual Technical Conference (USENIX ATC 2019, ERC), Innovations in Software Engineering Conference (ISEC 2020), The 18th Belgium-Netherlands Software Evolution, Workshop (BENEVOL 2019), 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2019), ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)
- Gilles Muller, Usenix ATC 2019, SOSP 2019, Supercomputing - system track 2019
- Pierre-Évariste Dagand: ICFP 2019, PLOS 2019

10.1.3. Journal

10.1.3.1. Member of the Editorial Boards

- Julia Lawall: Science of Computer Programming

10.1.3.2. Reviewer - Reviewing Activities

- Julia Lawall: IEEE Transactions on Software Engineering
- Pierre-Évariste Dagand: Journal of Functional Programming

10.1.4. Invited Talks

- Julia Lawall: Automation in the Maintenance of the Linux Kernel The Coccinelle Experience. Collège de France, February 6.

- Julia Lawall: Coccinelle: 10 Years of Automated Evolution in the Linux Kernel, keynote Linaro Connect conference, September 27.
- Julia Lawall: 10 Years of Automated Evolution in the Linux Kernel, keynote MASCOTS conference, October 23.
- Julia Lawall: Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers. Lund University, Tsinghua University, University of Illinois, National University of Singapore.
- Julia Lawall: Julia's adventures with Why3. CNAM.
- Lucas Serrano: SPINFER: Semantic patch inference for the Linux kernel. Singapore Management University.
- Gilles Muller: *Research problems in thread scheduling for multicore general purpose operating systems*, WOS Workshop (Rennes), University of Utah, University of New South Wales, University of Sydney, ENS Lyon.
- Pierre-Évariste Dagand: *Programming with dependent types*, Collège de France.

10.1.5. Scientific Expertise

Julia Lawall, Gilles Muller: Discussion with Starburst as part of their survey of sovereign operating systems (March 19, 2019).

10.1.6. Research Administration

- Gilles Muller: Member of a hiring committee for a Professor Position at ENS Lyon.
- Gilles Muller: Elected member of IFIP WG 10.4 (Dependability),
- Bertil Folliot: Elected member of the IFIP WG10.3 working group (Concurrent systems)
- Julia Lawall: Founding member of IFIP WG 2.11 (Program Generation),
- Pierre-Évariste Dagand: Member of a hiring committee for a "Professeur Assistant d'exercice incomplet en Informatique" at École Polytechnique

10.2. Teaching - Supervision - Juries

10.2.1. Teaching

- Professional Licence: Bertil Folliot, Programmation C, L2, SU, France
- Professional Licence: Bertil Folliot, Lab projects, L2, SU, France
- Licence: Pierre-Évariste Dagand, INF311: Introduction to Programming (40h), L1, École Polytechnique, France
- Licence: Pierre-Évariste Dagand, CSE205: Computer Architecture and Operating Systems (27h), Bachelor, École Polytechnique, France
- Master: Pierre-Évariste Dagand, MPRI 2-4.4: Dependently-typed programming (12h), M2, MPRI, France
- Master: Pierre-Évariste Dagand, INF559: Computer Architecture and Operating Systems (18h), M1, École Polytechnique, France

10.2.2. Supervision

- PhD in progress : Cédric Courtaud, CIFRE Thalès, 2016-2019 (defense January 28 2020), Gilles Muller, Julien Sopéna (Delys).
- PhD in progress : Redha Gouicem, 2016-2020, Gilles Muller, Julien Sopéna (Delys).
- PhD in progress : Yoann Ghigoff, 2019-2022, Gilles Muller, Julien Sopéna (Delys), Kahina Lazri (Orange Labs).

- PhD in progress : Darius Mercadier, 2017-2020, Pierre-Évariste Dagand, Gilles Muller.
- PhD in progress : Lucas Serrano, 2017-2020, Julia Lawall.
- PhD in progress : Pierre Nigron, 2019-2021, Pierre-Évariste Dagand, Julia Lawall.

10.2.3. Juries

- Julia Lawall: PhD jury of Jesper Öqvists (University of Lund, reporter), January 18, 2019.
- Julia Lawall: PhD jury of David Come (Onera, reporter), January 25, 2019.
- Julia Lawall: PhD jury of Jason Lecerf (University of Lille, reporter), November 26, 2019.
- Julia Lawall: HDR jury of Nic Volanschi (University of Bordeaux, president), November 29, 2019.
- Gilles Muller: PhD jury of Damien Carver (Sorbonne University, president), May 17, 2019.
- Pierre-Évariste Dagand: PhD jury of Kenji Maillard (University PSL, examiner), November 25, 2019.

10.3. Popularization

10.3.1. Internal or external Inria responsibilities

- Julia Lawall: Chair of the Commission des Emplois Scientifiques, with Mazyar Mirrahimi.

10.3.2. Education

- Julia Lawall: Coccinelle: Practical program transformation for the Linux kernel. Ecole de Jeunes Chercheurs en Programmation.

10.3.3. Interventions

- Julia Lawall: Systèmes d'exploitation. Girls Can Code.

11. Bibliography

Major publications by the team in recent years

[1] T. BOURKE, L. BRUN, P. DAGAND, X. LEROY, M. POUZET, L. RIEG. *A formally verified compiler for Lustre*, in "PLDI", 2017, pp. 586–601

[2] J. BRUNEL, D. DOLIGEZ, R. R. HANSEN, J. L. LAWALL, G. MULLER. *A foundation for flow-based program matching using temporal logic and model checking*, in "POPL", Savannah, GA, USA, ACM, January 2009, pp. 114–126

[3] L. BURGY, L. RÉVEILLÈRE, J. L. LAWALL, G. MULLER. *Zebu: A Language-Based Approach for Network Protocol Message Processing*, in "IEEE Trans. Software Eng.", 2011, vol. 37, n^{o} 4, pp. 575-591

[4] P. DAGAND, N. TABAREAU, É. TANTER. *Partial type equivalences for verified dependent interoperability*, in "ICFP", 2016, pp. 298–310

[5] D. MERCADIER, P. DAGAND. *Usuba: high-throughput and constant-time ciphers, by construction*, in "PLDI", 2019, pp. 157–173

[6] G. MULLER, C. CONSEL, R. MARLET, L. P. BARRETO, F. MÉRILLON, L. RÉVEILLÈRE. *Towards Robust OSes for Appliances: A New Approach Based on Domain-specific Languages*, in "Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System", Kolding, Denmark, 2000, pp. 19–24

[7] G. MULLER, J. L. LAWALL, H. DUCHESNE. *A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation*, in "HASE - High Assurance Systems Engineering Conference", Heidelberg, Germany, IEEE, October 2005, pp. 56–65

[8] F. MÉRILLON, L. RÉVEILLÈRE, C. CONSEL, R. MARLET, G. MULLER. *Devil: An IDL for hardware programming*, in "Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)", San Diego, California, USENIX Association, October 2000, pp. 17–30

[9] Y. PADIOLEAU, J. L. LAWALL, R. R. HANSEN, G. MULLER. *Documenting and Automating Collateral Evolutions in Linux Device Drivers*, in "EuroSys", Glasgow, Scotland, March 2008, pp. 247–260

[10] N. PALIX, G. THOMAS, S. SAHA, C. CALVÈS, J. L. LAWALL, G. MULLER. *Faults in Linux 2.6*, in "ACM Transactions on Computer Systems", June 2014, vol. 32, n^{o} 2, pp. 4:1–4:40

Publications of the year

Articles in International Peer-Reviewed Journals

[11] T. HOANG, J. L. LAWALL, Y. TIAN, R. J. OENTARYO, D. LO. *PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel*, in "IEEE Transactions on Software Engineering", November 2019, 17 p. [*DOI :* 10.1109/TSE.2019.2952614], https://hal.inria.fr/hal-02373994

[12] G. LENA COTA, S. BEN MOKHTAR, G. GIANINI, E. DAMIANI, J. L. LAWALL, G. MULLER, L. BRUNIE. *RACOON++: A Semi-Automatic Framework for the Selfishness-Aware Design of Cooperative Systems*, in "IEEE Transactions on Dependable and Secure Computing", July 2019, vol. 16, n^{o} 4, pp. 635-650 [*DOI :* 10.1109/TDSC.2017.2706286], https://hal.archives-ouvertes.fr/hal-02196805

International Conferences with Proceedings

[13] J.-J. BAI, J. L. LAWALL, Q.-L. CHEN, S.-M. HU. *Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers*, in "2019 USENIX Annual Technical Conference", Renton, Washington, United States, July 2019, https://hal.inria.fr/hal-02182516

[14] J.-J. BAI, J. L. LAWALL, W. TAN, S.-M. HU. *DCNS: Automated Detection of Conservative Non-Sleep Defects in the Linux Kernel*, in "ASPLOS 2019 - The 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems", Providence, Rhode Island, United States, ACM, April 2019, pp. 287-299 [*DOI :* 10.1145/3297858.3304065], https://hal.inria.fr/hal-02389543

[15] B. BUI, D. MVONDO, B. TEABE, K. JIOKENG, L. WAPET, A. TCHANA, G. THOMAS, D. HAGIMONT, G. MULLER, N. DEPALMA. *When eXtended Para-Virtualization (XPV) meets NUMA*, in "EUROSYS 2019: 14th European Conference on Computer Systems", Dresde, Germany, ACM Press, 2019, 7 p. [*DOI :* 10.1145/3302424.3303960], https://hal.archives-ouvertes.fr/hal-02333640

[16] D. CARVER, R. GOUICEM, J.-P. LOZI, J. SOPENA, B. LEPERS, W. ZWAENEPOEL, N. PALIX, J. L. LAWALL, G. MULLER. *Fork/Wait and Multicore Frequency Scaling: a Generational Clash*, in "10th Workshop on Programming Languages and Operating Systems", Huntsville, Canada, ACM Press, October 2019, pp. 53-59 [*DOI :* 10.1145/3365137.3365400], https://hal.inria.fr/hal-02349987

[17] Q.-L. CHEN, J.-J. BAI, Z.-M. JIANG, J. L. LAWALL, S.-M. HU. *Detecting Data Races Caused by Inconsistent Lock Protection in Device Drivers*, in "SANER 2019 - 26th IEEE International Conference on Software Analysis, Evolution and Reengineering", Hangzhou, China, February 2019, https://hal.inria.fr/hal-02014196

[18] C. COURTAUD, J. SOPENA, G. MULLER, D. GRACIA. *Improving Prediction Accuracy of Memory Interferences for Multicore Platforms*, in "RTSS 2019 - 40th IEEE Real-Time Systems Symposium", Hong-Kong, China, IEEE, December 2019, https://hal.inria.fr/hal-02401625

[19] T. HOANG, J. L. LAWALL, R. J. OENTARYO, Y. TIAN, D. LO. *PatchNet: A Tool for Deep Patch Classification*, in "ICSE-Companion 2019 - IEEE/ACM 41st International Conference on Software Engineering", Montreal, Canada, IEEE, May 2019, pp. 83-86 [*DOI :* 10.1109/ICSE-COMPANION.2019.00044], https://hal.inria.fr/hal-02408347

[20] Z.-M. JIANG, J.-J. BAI, J. L. LAWALL, S.-M. HU. *Fuzzing Error Handling Code in Device Drivers Based on Software Fault Injection*, in "ISSRE 2019 - The 30th International Symposium on Software Reliability Engineering", Berlin, Germany, October 2019 [*DOI :* 10.1109/ISSRE.2019.00022], https://hal.inria.fr/hal-02389293

[21] H. J. KANG, F. THUNG, J. L. LAWALL, G. MULLER, L. JIANG, D. LO. *Semantic Patches for Java Program Transformation*, in "33rd European Conference on Object-Oriented Programming (ECOOP 2019)", London, United Kingdom, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, July 2019, vol. 134, pp. 22:1–22:27 [*DOI :* 10.4230/LIPICS.ECOOP.2019.22], https://hal.inria.fr/hal-02182522

[22] F. LANIEL, D. CARVER, J. SOPENA, F. WAJSBURT, J. LEJEUNE, M. SHAPIRO. *Highlighting the Container Memory Consolidation Problems in Linux*, in "2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)", Cambridge, United States, IEEE, September 2019, pp. 1-4 [*DOI :* 10.1109/NCA.2019.8935034], https://hal.archives-ouvertes.fr/hal-02424007

[23] D. MERCADIER, P.-É. DAGAND. *Usuba: high-throughput and constant-time ciphers, by construction*, in "PLDI 2019 - 40th ACM SIGPLAN Conference on Programming Language Design and Implementation", Phoenix, United States, ACM Press, June 2019, pp. 157-173 [*DOI :* 10.1145/3314221.3314636], https://hal.inria.fr/hal-02176603

Research Reports

[24] H. J. KANG, F. THUNG, J. L. LAWALL, G. MULLER, L. JIANG, D. LO. *Automating Program Transformation for Java Using Semantic Patches*, Inria Paris, February 2019, n° RR-9256, https://hal.inria.fr/hal-02023368

References in notes

[25] T. BALL, E. BOUNIMOVA, B. COOK, V. LEVIN, J. LICHTENBERG, C. MCGARVEY, B. ONDRUSEK, S. K. RAJAMANI, A. USTUNER. *Thorough Static Analysis of Device Drivers*, in "EuroSys", 2006, pp. 73–85

[26] A. BAUMANN, P. BARHAM, P.-É. DAGAND, T. HARRIS, R. ISAACS, S. PETER, T. ROSCOE, A. SCHÜPBACH, A. SINGHANIA. *The multikernel: A new OS architecture for scalable multicore systems*, in "SOSP", 2009, pp. 29–44

[27] T. F. BISSYANDÉ, L. RÉVEILLÈRE, J. L. LAWALL, Y.-D. BROMBERG, G. MULLER. *Implementing an embedded compiler using program transformation rules*, in "Software: Practice and Experience", 2013

[28] T. F. BISSYANDÉ, L. RÉVEILLÈRE, J. L. LAWALL, Y.-D. BROMBERG, G. MULLER. *Implementing an Embedded Compiler using Program Transformation Rules*, in "Software: Practice and Experience", February 2015, vol. 45, n^{o} 2, pp. 177-196, https://hal.archives-ouvertes.fr/hal-00844536

[29] T. F. BISSYANDÉ, L. RÉVEILLÈRE, J. L. LAWALL, G. MULLER. *Ahead of Time Static Analysis for Automatic Generation of Debugging Interfaces to the Linux Kernel*, in "Automated Software Engineering", May 2014, pp. 1-39 [*DOI :* 10.1007/s10515-014-0152-4], https://hal.archives-ouvertes.fr/hal-00992283

[30] A. P. BLACK, S. DUCASSE, O. NIERSTRASZ, D. POLLET. *Pharo by Example*, Square Bracket Associates, 2010

[31] E. BRADY, K. HAMMOND. *Resource-Safe Systems Programming with Embedded Domain Specific Languages*, in "14th International Symposium on Practical Aspects of Declarative Languages (PADL)", LNCS, Springer, 2012, vol. 7149, pp. 242–257

[32] T. BRAIBANT, D. POUS. *An Efficient Coq Tactic for Deciding Kleene Algebras*, in "1st International Conference on Interactive Theorem Proving (ITP)", LNCS, Springer, 2010, vol. 6172, pp. 163–178

[33] C. CADAR, D. DUNBAR, D. R. ENGLER. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, in "OSDI", 2008, pp. 209–224

[34] V. CHIPOUNOV, G. CANDEA. *Reverse Engineering of Binary Device Drivers with RevNIC*, in "EuroSys", 2010, pp. 167–180

[35] A. CHLIPALA. *The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier*, in "ICFP", 2013, pp. 391–402

[36] L. A. CLARKE. *A system to generate test data and symbolically execute programs*, in "IEEE Transactions on Software Engineering", 1976, vol. 2, n^{o} 3, pp. 215–222

[37] E. CLARKE, O. GRUMBERG, S. JHA, Y. LU, H. VEITH. *Counterexample-guided abstraction refinement for symbolic model checking*, in "J. ACM", 2003, vol. 50, n^{o} 5, pp. 752–794

[38] P. COUSOT, R. COUSOT. *Abstract Interpretation: Past, Present and Future*, in "CSL-LICS", 2014, pp. 2:1–2:10

[39] P.-É. DAGAND, A. BAUMANN, T. ROSCOE. *Filet-o-Fish: practical and dependable domain-specific languages for OS development*, in "Programming Languages and Operating Systems (PLOS)", 2009, pp. 51–55

[40] I. DILLIG, T. DILLIG, A. AIKEN. *Sound, complete and scalable path-sensitive analysis*, in "PLDI", June 2008, pp. 270–280

[41] D. R. ENGLER, B. CHELF, A. CHOU, S. HALLEM. *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, in "OSDI", 2000, pp. 1–16

[42] D. R. ENGLER, D. Y. CHEN, A. CHOU, B. CHELF. *Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code*, in "SOSP", 2001, pp. 57–72

[43] A. GOLDBERG, D. ROBSON. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983

[44] L. GU, A. VAYNBERG, B. FORD, Z. SHAO, D. COSTANZO. *CertiKOS: A Certified Kernel for Secure Cloud Computing*, in "Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)", 2011, pp. 3:1–3:5

[45] L. GUO, J. L. LAWALL, G. MULLER. *Oops! Where did that code snippet come from?*, in "11th Working Conference on Mining Software Repositories, MSR", Hyderabad, India, ACM, May 2014, pp. 52–61

[46] A. ISRAELI, D. G. FEITELSON. *The Linux kernel as a case study in software evolution*, in "Journal of Systems and Software", 2010, vol. 83, n^{o} 3, pp. 485–501

[47] A. KADAV, M. M. SWIFT. *Understanding modern device drivers*, in "ASPLOS", 2012, pp. 87–98

[48] A. KENNEDY, N. BENTON, J. B. JENSEN, P.-É. DAGAND. *Coq: The World's Best Macro Assembler?*, in "PPDP", Madrid, Spain, ACM, 2013, pp. 13–24

[49] G. A. KILDALL. *A Unified Approach to Global Program Optimization*, in "POPL", 1973, pp. 194–206

[50] G. KLEIN, K. ELPHINSTONE, G. HEISER, J. ANDRONICK, D. COCK, P. DERRIN, D. ELKADUWE, K. ENGELHARDT, R. KOLANSKI, M. NORRISH, T. SEWELL, H. TUCH, S. WINWOOD. *seL4: formal verification of an OS kernel*, in "SOSP", 2009, pp. 207–220

[51] J. L. LAWALL, J. BRUNEL, N. PALIX, R. R. HANSEN, H. STUART, G. MULLER. *WYSIWIB: Exploiting fine-grained program structure in a scriptable API-usage protocol-finding process*, in "Software, Practice Experience", 2013, vol. 43, n^{o} 1, pp. 67–92

[52] J. L. LAWALL, B. LAURIE, R. R. HANSEN, N. PALIX, G. MULLER. *Finding Error Handling Bugs in OpenSSL using Coccinelle*, in "Proceeding of the 8th European Dependable Computing Conference (EDCC)", Valencia, Spain, April 2010, pp. 191–196

[53] J. L. LAWALL, D. LO. *An automated approach for finding variable-constant pairing bugs*, in "25th IEEE/ACM International Conference on Automated Software Engineering", Antwerp, Belgium, September 2010, pp. 103–112

[54] J. L. LAWALL, D. PALINSKI, L. GNIRKE, G. MULLER. *Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers*, in "2017 USENIX Annual Technical Conference", Santa Clara, CA, United States, July 2017, 12 p. , https://hal.inria.fr/hal-01556589

[55] C. LE GOUES, W. WEIMER. *Specification Mining with Few False Positives*, in "TACAS", York, UK, Lecture Notes in Computer Science, March 2009, vol. 5505, pp. 292–306

[56] Z. LI, S. LU, S. MYAGMAR, Y. ZHOU. *CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code*, in "OSDI", 2004, pp. 289–302

[57] Z. LI, Y. ZHOU. *PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code*, in "Proceedings of the 10th European Software Engineering Conference", 2005, pp. 306–315

[58] D. LO, S. KHOO. *SMArTIC: towards building an accurate, robust and scalable specification miner*, in "FSE", 2006, pp. 265–275

[59] J.-P. LOZI, F. DAVID, G. THOMAS, J. L. LAWALL, G. MULLER. *Fast and Portable Locking for Multicore Architectures*, in "ACM Transactions on Computer Systems", January 2016 [*DOI :* 10.1145/2845079], https://hal.inria.fr/hal-01252167

[60] S. LU, S. PARK, Y. ZHOU. *Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing*, in "IEEE Transactions on Software Engineering", 2012, vol. 38, n^{o} 4, pp. 844–860

[61] M. MERNIK, J. HEERING, A. M. SLOANE. *When and How to Develop Domain-specific Languages*, in "ACM Comput. Surv.", December 2005, vol. 37, n^{o} 4, pp. 316–344, http://dx.doi.org/10.1145/1118890.1118892

[62] G. MORRISETT, G. TAN, J. TASSAROTTI, J.-B. TRISTAN, E. GAN. *RockSalt: better, faster, stronger SFI for the x86*, in "PLDI", 2012, pp. 395-404

[63] M. ODERSKY, T. ROMPF. *Unifying functional and object-oriented programming with Scala*, in "Commun. ACM", 2014, vol. 57, n^{o} 4, pp. 76–86

[64] M. C. OLESEN, R. R. HANSEN, J. L. LAWALL, N. PALIX. *Coccinelle: Tool support for automated CERT C Secure Coding Standard certification*, in "Science of Computer Programming", October 2014, vol. 91, n^{o} B, pp. 141–160, https://hal.inria.fr/hal-01096185

[65] T. REPS, T. BALL, M. DAS, J. LARUS. *The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem*, in "ESEC/FSE", 1997, pp. 432–449

[66] L. R. RODRIGUEZ, J. L. LAWALL. *Increasing Automation in the Backporting of Linux Drivers Using Coccinelle*, in "11th European Dependable Computing Conference - Dependability in Practice", Paris, France, 11th European Dependable Computing Conference - Dependability in Practice, November 2015, https://hal.inria.fr/hal-01213912

[67] C. RUBIO-GONZÁLEZ, H. S. GUNAWI, B. LIBLIT, R. H. ARPACI-DUSSEAU, A. C. ARPACI-DUSSEAU. *Error propagation analysis for file systems*, in "PLDI", Dublin, Ireland, ACM, June 2009, pp. 270–280

[68] L. RYZHYK, P. CHUBB, I. KUZ, E. LE SUEUR, G. HEISER. *Automatic device driver synthesis with Termite*, in "SOSP", 2009, pp. 73–86

[69] L. RYZHYK, A. WALKER, J. KEYS, A. LEGG, A. RAGHUNATH, M. STUMM, M. VIJ. *User-Guided Device Driver Synthesis*, in "OSDI", 2014, pp. 661–676

[70] R. K. SAHA, J. L. LAWALL, S. KHURSHID, D. E. PERRY. *On the Effectiveness of Information Retrieval Based Bug Localization for C Programs*, in "ICSME 2014 - 30th International Conference on Software Maintenance and Evolution", Victoria, Canada, IEEE, September 2014, pp. 161-170 [*DOI :* 10.1109/ICSME.2014.38], https://hal.inria.fr/hal-01086082

[71] R. SAHA, J. L. LAWALL, S. KHURSHID, D. E. PERRY. *On the Effectiveness of Information Retrieval based Bug Localization for C Programs*, in "International Conference on Software Maintenance and Evolution (ICSME)", Victoria, BC, Canada, September 2014

[72] S. SAHA, J.-P. LOZI, G. THOMAS, J. L. LAWALL, G. MULLER. *Hector: Detecting resource-release omission faults in error-handling code for systems software*, in "DSN 2013 - 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)", Budapest, Hungary, IEEE Computer Society, June 2013, pp. 1-12 [*DOI :* 10.1109/DSN.2013.6575307], https://hal.inria.fr/hal-00918079

[73] D. A. SCHMIDT. *Data Flow Analysis is Model Checking of Abstract Interpretations*, in "POPL", 1998, pp. 38–48

[74] P. SENNA TSCHUDIN, J. L. LAWALL, G. MULLER. *3L: Learning Linux Logging*, in "BElgian-NEtherlands software eVOLution seminar (BENEVOL 2015)", Lille, France, December 2015, https://hal.inria.fr/hal-01239980

[75] M. SHAPIRO. *Purpose-built languages*, in "Commun. ACM", 2009, vol. 52, n^{o} 4, pp. 36–41

[76] R. TARTLER, D. LOHMANN, J. SINCERO, W. SCHRÖDER-PREIKSCHAT. *Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem*, in "EuroSys", 2011, pp. 47–60

[77] F. THUNG, D. X. B. LE, D. LO, J. L. LAWALL. *Recommending Code Changes for Automatic Backporting of Linux Device Drivers*, in "32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)", Raleigh, North Carolina, United States, IEEE, October 2016, https://hal.inria.fr/hal-01355859

[78] W. WANG, M. GODFREY. *A Study of Cloning in the Linux SCSI Drivers*, in "Source Code Analysis and Manipulation (SCAM)", IEEE, 2011

[79] J. YANG, C. HAWBLITZEL. *Safe to the Last Instruction: Automated Verification of a Type-safe Operating System*, in "PLDI", 2010, pp. 99–110